Nogood-Based Asynchronous Forward Checking Algorithms

Mohamed Wahbi, Redouane Ezzahir, Christian Bessiere, El Houssine Bouyakhf

Nogood-Based Asynchronous Forward Checking Algorithms

Mohamed Wahbi *
TASC INRIA, LINA CNRS, Ecole des Mines de Nantes, France
mohamed.wahbi@mines-nantes.fr

Redouane Ezzahir
ENSA Agadir, University Ibn Zohr, Morocco
red.ezzahir@gmail.com

Christian Bessiere
CNRS, University of Montpellier, France
bessiere@lirmm.fr

El Houssine Bouyakhf
LIMIARF, University Mohammed V–Agdal, Morocco
bouyakhf@fsr.ac.ma

Abstract

We propose two new algorithms for solving Distributed Constraint Satisfaction Problems (DisCSPs). The first algorithm, AFC-ng, is a nogood-based version of Asynchronous Forward Checking (AFC). Besides its use of nogoods as justification of value removals, AFC-ng allows simultaneous backtracks going from different agents to different destinations. The second algorithm, Asynchronous Forward Checking Tree (AFC-tree), is based on the AFC-ng algorithm and is performed on a pseudo-tree ordering of the constraint graph. AFC-tree runs simultaneous search processes in disjoint problem subtrees and exploits the parallelism inherent in the problem. We prove that AFC-ng and AFC-tree only need polynomial space. We compare the performance of these algorithms with other DisCSP algorithms on random DisCSPs and instances from real benchmarks: sensor networks and distributed meeting scheduling. Our experiments show that AFC-ng improves on AFC and that AFC-tree outperforms all compared algorithms, particularly on sparse problems.

1 Introduction

1.1 Context

Constraint programming is an area in computer science that has gained increasing interest in recent years. Constraint programming is based on its powerful framework named *Constraint*

*Parts of this work were carried out while the first author was a PhD student at University of Montpellier and University Mohammed V–Agdal.

Satisfaction Problem (CSP). CSP is a general framework that can formalize many real world combinatorial problems such as resource allocation, car sequencing, natural language understanding, machine vision, etc. A constraint satisfaction problem consists in looking for solutions to a constraint network, that is, a set of assignments of values to variables that satisfy the constraints of the problem. These constraints represent restrictions on values combinations allowed for constrained variables.

There exist applications that are of a distributed nature. In this kind of applications the knowledge about the problem, that is, variables and constraints, is distributed among physical distributed agents. This distribution is mainly due to privacy and/or security requirements: constraints or possible values may be strategic information that should not be revealed to other agents that can be seen as competitors. Several applications in multi-agent coordination are of such kind. Examples of applications are Sensor Networks [20, 2], Military Unmanned Aerial Vehicles Teams [20], Distributed Meeting Scheduling Problems [36, 23], Distributed Resource Allocation Problems [29], Log-based Reconciliation [11], Distributed Vehicle Routing Problems [21], etc. Therefore, the distributed framework *Distributed Constraint Satisfaction Problems* (DisCSP) is used to model and solve this kind of problems.

A DisCSP is composed of a group of autonomous agents, where each agent has control of some elements of information about the whole problem, that is, variables and constraints. Each agent owns its local constraint network. Variables in different agents are connected by constraints. Agents must assign values to their variables so that all constraints are satisfied. Hence, agents assign values to their variables, attempting to generate a locally consistent assignment that is also consistent with constraints between agents [40, 38]. To achieve this goal, agents check the value assignments to their variables for local consistency and exchange messages to check consistency of their proposed assignments against constraints that contain variables that belong to other agents.

1.2 Related work

Several distributed algorithms for solving DisCSPs have been developed in the last two decades. The first complete asynchronous search algorithm for solving DisCSPs is Asynchronous Backtracking (ABT) [39, 38, 4]. ABT is an asynchronous algorithm executed autonomously by each agent in the distributed problem. Synchronous Backtrack (SBT) is the simplest DisCSP search algorithm. SBT performs assignments sequentially and synchronously. SBT agents assign their variables one by one, recording their assignments on a data structure called the Current Partial Assignment (CPA). In SBT, only the agent holding the CPA performs an assignment or backtracks [41]. Brito and Meseguer (2008) proposed ABT-hyb, a hybrid algorithm that adds synchronization points in ABT so as to avoid some redundant messages after backtracking. They experimentally show that in most instances ABT-hyb outperforms ABT in terms of computation effort and communication load.

Extending SBT, Meisels and Zivan (2007) proposed the Asynchronous Forward Checking (AFC). Besides assigning variables sequentially as is done in SBT, agents in AFC perform forward checking (FC [18]) asynchronously. The key here is that each time an agent succeeds to extend the current partial assignment (by assigning its variable), it sends the CPA to its successor and sends copies of this CPA to all agents connected to itself whose assignments are not yet on the CPA. When an agent receives a copy of the CPA, it performs the forward checking phase. In the forward checking phase all inconsistent values with assignments on the received CPA are removed. The forward checking operation is performed asynchronously

–from where comes the name of the algorithm. When an agent generates an empty domain as a result of a forward checking, it informs agents with unassigned variables on the (inconsistent) CPA. Afterwards, only the agent that receives the CPA from its predecessor and is holding the inconsistent CPA will eventually backtrack. Hence, in AFC, backtracking is done sequentially, and at any time there is only either one CPA or one backtrack message being sent in the network. Meisels and Zivan have shown in [26] that AFC is computationally more efficient than ABT. However, due to the manner in which the backtrack operation is performed, AFC does not draw all the benefit it could from the asynchronism of the FC phase.

In [28], Nguyen et al. have proposed Distributed BackJumping (DBJ), an improved version of the basic AFC that addresses its backtrack operation. In DBJ, the agent who detects the empty domain can itself perform the backtrack operation by backjumping directly to the culprit agent. It sends a backtrack message to the last agent assigned in the inconsistent CPA. The agent who receives a backtrack message generates a new CPA that will dominate older ones thanks to a timestamp mechanism. DBJ still sends the inconsistent CPA to unassigned agents on it. DBJ does not use nogoods for justification of value removals. Consequently, DBJ only mimics the simple *BackJumping* (BJ) [16] although the authors report performing the Graph-based BackJumping (GBJ) [13].[1] Section 3.2 illustrates on an example that DBJ does not perform GBJ but just BJ. In the same work, Nguyen et al. presented the Dynamic Distributed BackJumping (DDBJ) algorithm. DDBJ is an improvement of DBJ that integrates heuristics for dynamic variable and value ordering, called the *possible conflict heuristics*. However, DDBJ requires additional messages to compute the dynamic ordering heuristics.

In [15], Freuder and Quinn introduced the concept of pseudo-tree, an efficient structure for solving centralized CSPs. Based on a "divide and conquer" principle provided by the pseudo-tree, they perform search in parallel. Depth-First Search trees (DFS-trees) are special cases of pseudo-trees. They are used in the *Network Consistency Protocol* (NCP) proposed in [12] by Collin et al.. In NCP, agents are prioritized using a DFS-tree. Agents on the same branch of the DFS-tree act synchronously, but agents having the same parent can act concurrently. A number of other algorithms for Distributed Constraint Optimization (DCOP) use pseudo-tree or DFS-tree orderings of the agents [27, 30, 31, 9, 37].

1.3 Our contribution

In this work, we present two new algorithms for solving DisCSPs. The first one is based on Asynchronous Forward Checking (AFC) and uses nogood as justifications of value removals. We call it Nogood-Based Asynchronous Forward Checking (AFC-ng). Unlike AFC, AFC-ng allows concurrent backtracks to be performed at the same time coming from different agents having an empty domain to different destinations. As a result, several CPAs could be generated simultaneously by the destination agents. Thanks to the timestamps integrated in the CPAs, the *strongest* CPA coming from the highest level in the agent ordering will eventually dominate all others. Interestingly, the search process with the strongest CPA will benefit from the computational effort done by the (killed) lower level processes. This is done by taking advantage from nogoods recorded when processing these lower level processes.

The second algorithm we propose is based on AFC-ng and is named Asynchronous

[1] BJ cannot execute two 'jumps' in a row, only performing steps back after a jump, whereas GBJ can perform sequences of consecutive jumps.

Forward Checking Tree (AFC-tree). The main feature of the AFC-tree algorithm is using different agents to search non-intersecting parts of the search space concurrently. In AFC-tree, agents are prioritized according to a pseudo-tree arrangement of the constraint graph. The pseudo-tree ordering is build in a preprocessing step. Using this priority ordering, AFC-tree performs multiple AFC-ng processes on the paths from the root to the leaves of the pseudo-tree. The agents that are brothers are committed to concurrently find the partial solutions of their variables. Therefore, AFC-tree exploits the potential speed-up of a parallel exploration in the processing of distributed problems [15]. A solution is found when all leaf agents succeed in extending the CPA they received. Furthermore, in AFC-tree privacy may be enhanced because communication is restricted to agents in the same branch of the pseudo-tree.

This paper is organized as follows. Section 2 gives the necessary background on DisCSPs and on the AFC algorithm. Sections 3 and 4 describe the algorithms AFC-ng and AFC-tree. Correctness proofs are given in Section 5. Section 6 presents an experimental evaluation of our proposed algorithms against other well-known distributed algorithms and we conclude the paper in Section 7.

2 Background

2.1 Basic definitions and notations

The *distributed constraint satisfaction problem* (DisCSP) has been formalized in [40] as a tuple $(\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{A}$ is a set of a agents $\{A_1, \ldots, A_a\}$, $\mathcal{X}$ is a set of n variables $\{x_1, \ldots, x_n\}$, where each variable x_i is controlled by one agent in $\mathcal{A}$. $\mathcal{D}^0 = \{D^0(x_1), \ldots, D^0(x_n)\}$ is a set of n domains, where $D^0(x_i)$ is the initial set of possible values to which variable x_i may be assigned. During search, values may be pruned from the domain. At any node, the set of possible values for variable x_i is denoted by $D(x_i)$ and is called the current domain of x_i. Only the agent who is assigned a variable has control on its value and knowledge of its domain. $\mathcal{C}$ is a set of constraints that specify the combinations of values allowed for the variables they involve. In this paper, we assume a binary distributed constraint network where all constraints are binary constraints (they involve two variables). A constraint $c_{ij} \in \mathcal{C}$ between two variables x_i and x_j is a subset of the Cartesian product of their domains ($c_{ij} \subseteq D^0(x_i) \times D^0(x_j)$). When there exists a constraint between two variables x_i and x_j, these variables are called ***neighbors***. The set of neighbors of a variable x_i is denoted by $\Gamma(x_i)$. The connectivity between the variables can be represented with a constraint graph G, where vertexes represent the variables and edges represent the constraints [14].

For simplicity purposes, we consider a restricted version of DisCSP where each agent controls exactly one variable ($a = n$). Thus, we use the terms agent and variable interchangeably and we identify the agent ID with its variable index. Furthermore, all agents store a unique total order $\prec$ on agents. Agents appearing before an agent $A_i \in \mathcal{A}$ in the total order are the higher agents (predecessors) and agents appearing after A_i are the lower agents (successors). The order $\prec$ divides the set $\Gamma(x_i)$ of neighbors of A_i into higher priority neighbors $\Gamma^-(x_i)$, and lower priority neighbors $\Gamma^+(x_i)$. For sake of clarity, we assume that the total order is the lexicographic ordering $[A_1, A_2, \ldots, A_n]$. Each agent maintains a counter, and increments it whenever it changes its value. The current value of the counter *tags* each generated assignment.

Definition 1. *An **assignment** for an agent $A_i \in \mathcal{A}$ is a tuple (x_i, v_i, t_i), where v_i is a value from the domain of x_i and t_i is the tag value. When comparing two assignments, the most up to date is the one with the greatest tag t_i.*

Definition 2. *A **current partial assignment** CPA is an ordered set of assignments $\{[(x_1, v_1, t_1), \ldots, (x_i, v_i, t_i)] \mid x_1 \prec \cdots \prec x_i\}$. Two CPAs are **Compatible** if they do not disagree on any variable value.*

Definition 3. *A **timestamp** associated with a CPA is an ordered list of counters $[t_1, t_2, \ldots, t_i]$ where t_j is the tag of the variable x_j. When comparing two CPAs, the **strongest** one is that associated with the lexicographically greater timestamp. That is, the CPA with greatest value on the first counter on which they differ, if any, otherwise the longest one.*

Definition 4. *The **AgentView** of an agent $A_i \in \mathcal{A}$ stores the most up to date assignments received from higher priority agents in the agent ordering. It has a form similar to a CPA and is initialized to the set of empty assignments.*

During search agents can infer inconsistent sets of assignments called nogoods.

Definition 5. *A **nogood** ruling out value v_k from the initial domain of variable x_k is a clause of the form $x_i = v_i \wedge x_j = v_j \wedge \ldots \rightarrow x_k \neq v_k$, meaning that the assignment $x_k = v_k$ is inconsistent with the assignments $x_i = v_i, x_j = v_j, \ldots$. The* left hand side *($lhs$) and the* right hand side *(rhs) are defined from the position of $\rightarrow$.*

The current domain $D(x_i)$ of a variable x_i contains all values from the initial domain $D^0(x_i)$ that are not ruled out by a nogood. When all values of a variable x_k are ruled out by some nogoods ($D(x_i) = \emptyset$), these nogoods are resolved, producing a new nogood (ng). Let x_j be the lowest variable in the left-hand side of all these nogoods and $x_j = v_j$. The $lhs(ng)$ is the conjunction of the left-hand sides of all nogoods except $x_j = v_j$ and $rhs(ng)$ is $x_j \neq v_j$.

2.2 Asynchronous Forward Checking

Asynchronous Forward Checking (AFC) incorporates the idea of the forward checking (FC) algorithm for centralized CSP [18]. However, agents perform the forward checking phase asynchronously [25, 26]. As in synchronous backtracking, agents assign their variables only when they hold the current partial assignment (***cpa***). The ***cpa*** is a unique message (token) that is passed from one agent to the next one in the ordering. The ***cpa*** message carries the current partial assignment (CPA) that agents attempt to extend into a complete solution by assigning their variables on it. When an agent succeeds in assigning its variable on the CPA, it sends this CPA to its successor. Furthermore, copies of the CPA are sent to all connected agents whose assignments are not yet on the CPA. These agents perform the forward checking asynchronously in order to detect as early as possible inconsistent partial assignments. The forward checking process is performed as follows. When an agent receives a CPA, it updates the domain of its variable, removing all values that are in conflict with assignments on the received CPA. Furthermore, the shortest CPA producing the inconsistency is stored as justification of the value deletion.

When an agent generates an empty domain as a result of a forward checking, it initiates a backtrack process by sending ***not_ok*** messages. ***not_ok*** messages carry the shortest inconsistent partial assignment (CPA) which caused the empty domain. ***not_ok*** messages

are sent to all agents with unassigned variables on the (inconsistent) CPA. When an agent receives the ***not_ok*** message, it checks if the CPA carried in the received message is compatible with its AgentView. If it is the case, the receiver stores the ***not_ok***, otherwise, the ***not_ok*** is discarded. When an agent holding a ***not_ok*** receives a CPA on a ***cpa*** message from its predecessor, it sends this ***not_ok*** back in a ***backcpa*** message. When multiple agents reject a given assignment by sending ***not_ok*** messages, only the first agent that will receive a ***cpa*** message from its predecessor and is holding a relevant ***not_ok*** message will eventually backtrack. After receiving a new ***cpa*** message, the ***not_ok*** message becomes obsolete when the CPA it carries is no longer a subset of the received CPA.

The manner in which the backtrack operation is performed is a major drawback of the AFC algorithm. The backtrack operation requires a lot of work from the agents. In addition, the backtrack is performed synchronously, and at any time, there is only either one ***cpa*** or one ***backcpa*** message being sent in the network.

3 Nogood-based Asynchronous Forward Checking

The nogood-based Asynchronous Forward Checking (AFC-ng) is based on AFC. AFC-ng tries to enhance the asynchronism of the forward checking phase. The two main features of AFC-ng are the following. First, it uses the nogoods as justification of value deletions. Each time an agent performs a forward check, it revises its *initial domain*, (including values already removed by a stored nogood) in order to store the best nogoods for removed values (one nogood per value). When comparing two nogoods eliminating the same value, the nogood with the *highest possible lowest variable* involved is selected (HPLV heuristic) [19]. When an empty domain is found, the resolved nogood contains variables as high as possible in the ordering, so that the backtrack message is sent as high as possible, thus saving unnecessary search effort [4]. By this ability to backtrack directly to agents responsible of the inconsistency, AFC-ng mimics the conflict-directed backjumping CBJ mechanism proposed for centralized CSPs [32].

Second, each time an agent A_i generates an empty domain it no longer sends ***not_ok*** messages. It resolves the nogoods ruling out values from its domain, producing a new nogood ng. ng is the conjunction of lhs of all nogoods stored by A_i. Then, A_i sends the resolved nogood ng in a ***ngd*** (backtrack) message to the lowest agent in ng. Hence, multiple backtracks may be performed at the same time coming from different agents having an empty domain. These backtracks are sent concurrently by these different agents to different destinations. The reassignment of the destination agents then happen simultaneously and generate several CPAs. However, the strongest CPA coming from the highest level in the agent ordering will eventually dominate all others. Agents use the timestamp (see Definition 3) to detect the strongest CPA. Interestingly, the search process of higher levels with stronger CPAs can use nogoods reported by the (killed) lower level processes, so that it benefits from their computational effort.

3.1 Description of the algorithm

The pseudo code of AFC-ng, executed by each agent, say A_i, is shown in Figures 1 and 2. Agent A_i stores a nogood per removed value in the NogoodStore. The NogoodStore of A_i is a set of nogoods that are compatible with the AgentView of A_i and that each justify the removal of a value from $D^0(x_i) \setminus D(x_i)$. The other values not ruled out by a nogood are in

$D(x_i)$, the current domain of x_i. In the following, v_i will represent the current value assigned to x_i and t_i the counter tagging v_i. t_i is used for the timestamp mechanism. "$v_i \leftarrow empty$" means that x_i is unassigned.

```
procedure AFC-ng()
01.  InitAgentView() ;
02.  end ← false; AgentView.Consistent ← true;
03.  if ( A_i = IA ) then Assign();
04.  while ( ¬end ) do
05.      msg ← getMsg();
06.      switch ( msg.type ) do
07.          cpa     : ProcessCPA(msg);
08.          ngd     : ProcessNogood(msg);
09.          stp     : end ← true ;

procedure InitAgentView()
10.  foreach ( x_j ≺ x_i ) do  AgentView[j] ← {(x_j, empty, 0)} ;

procedure Assign()
11.  if ( D(x_i) ≠ ∅ ) then
12.      v_i ← ChooseValue() ; t_i ← t_i+1 ;
13.      CPA ← {AgentView ∪ (x_i, v_i, t_i)} ;
14.      SendCPA(CPA) ;
15.  else Backtrack();

procedure SendCPA(CPA)
16.  if ( size(CPA) = n ) then          /* A_i is the last agent in the total ordering */
17.      Report SOLUTION;     end ← true ;
18.  else foreach ( x_k ∈ Γ+(x_i) ) do  sendMsg:cpa ⟨ CPA ⟩  to A_k ;

procedure ProcessCPA(msg)
19.  if ( msg.CPA is stronger than AgentView ) ) then
20.      UpdateAgentView(msg.CPA) ;
21.      AgentView.Consistent ← true ;
22.      Revise() ;
23.      if ( D(x_i) = ∅ ) then Backtrack() ;
24.      else CheckAssign(msg.Sender) ;

procedure CheckAssign(sender)
25.  if ( predecessor(A_i) = sender ) then Assign() ;
```

Figure 1: Nogood-based AFC algorithm running by agent A_i (Part 1).

Agent A_i starts the search by calling procedure `AFC-ng()` in which it initializes its AgentView (line 1) by setting counters to zero (line 10). The AgentView contains a consistency flag that represents whether the partial assignment it holds is consistent. If A_i is the initializing agent IA (the first agent in the agent ordering), it initiates the search by calling

procedure `Assign()` (line 3). Then, a loop considers the reception and the processing of the possible message types.

When calling `Assign()`, A_i looks for a value which is consistent with its AgentView. If A_i fails to find a consistent value, it calls procedure `Backtrack()` (line 15). If A_i succeeds, it increments its counter t_i and generates a CPA from its AgentView augmented by its assignment (line 13). Afterwards, A_i calls procedure `SendCPA(CPA)` (line 14). If the CPA includes all agents assignments (A_i is the lowest agent in the order, line 16), A_i reports the CPA as a solution of the problem and marks the *end* flag true to stop the main loop (lines 17-17). Otherwise, A_i sends forward the CPA to every connected agent whose assignments are not yet on the CPA (i.e., lower neighbors, line 18). So, the next agent on the ordering (successor) will try to extend this CPA by assigning its variable on it while other agents will perform the forward checking phase asynchronously to check its consistency.

Whenever A_i receives a ***cpa*** message, procedure `ProcessCPA` is called (line 7). If the received CPA is stronger than its AgentView, A_i updates its AgentView (`UpdateAgentView` call, line 20) and marks it consistent (line 21). Procedure `UpdateAgentView` (lines 39-41) sets the AgentView and the NogoodStore to be compatible with the received CPA. Each nogood in the NogoodStore containing a value for a variable different from that on the received CPA will be deleted (line 41). Next, A_i calls procedure `Revise` (line 22) to store nogoods for values inconsistent with the new AgentView or to try to find a better nogood for values already having one in the NogoodStore (line 44). A nogood is better according to the *HPLV* heuristic if the lowest variable in the body (*lhs*) of the nogood is higher. If A_i generates an empty domain as a result of calling `Revise`, it calls procedure `Backtrack` (line 23), otherwise, A_i calls procedure `CheckAssign` to check if it has to assign its variable (line 24). In `CheckAssign(`*sender*`)`, A_i calls procedure `Assign` to try to assign its variable only if sender is the predecessor of A_i (i.e., CPA was received from the predecessor, line 25).

When every value of A_i's variable is ruled out by a nogood (line 23), the procedure `Backtrack` is called. These nogoods are resolved by computing a new nogood ng (line 26). ng is the conjunction of the left hand sides of all nogoods stored by A_i in its NogoodStore. If the new nogood ng is empty, A_i reports the failure (the problem is unsolvable) and terminates the execution by setting *end* flag true (lines 27-28). Otherwise, A_i updates its AgentView by removing assignments of every agent that is placed after the agent A_j owner of `rhs(`ng`)` in the total order (lines 30-30). A_i also updates its NogoodStore by removing obsolete nogoods (line 33). Obsolete nogoods are nogoods incompatible with the AgentView or containing the assignment of x_j (the right hand side of ng) (line 32). Finally, A_i marks its AgentView as inconsistent, removes its last assignment (line 34) and it backtracks by sending a ***ngd*** message to agent A_j (the right hand side of ng) (line 35). The ***ngd*** message carries the generated nogood (ng). A_i remains in an inconsistent state until receiving a stronger CPA holding at least one agent assignment with counter higher than that in the AgentView of A_i.

When a ***ngd*** message is received by an agent A_i, it checks the validity of the received nogood (line 36). If the received nogood is compatible with the AgentView, this nogood is a valid justification for removing the value on its rhs. Then if the value on the rhs of the received nogood is already removed, A_i adds the received nogood to its NogoodStore if it is better (according to the HPLV heuristic) than the current stored nogood. If the value on the rhs of the received nogood belongs to the current domain of x_i, A_i simply adds it to its NogoodStore. If the value on the rhs of the received nogood equals v_i, the current value of A_i, A_i unassigns its variable and calls the procedure `Assign` (line 38).

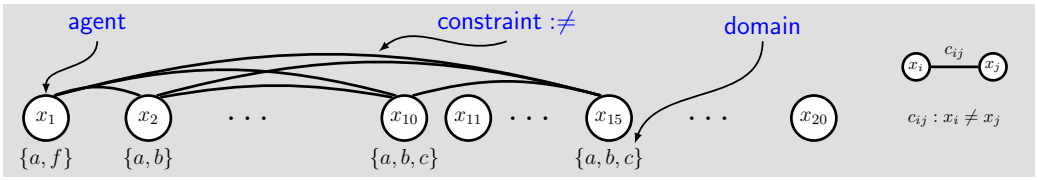
```
procedure Backtrack()
26.  ng ← solve(NogoodStore) ;
27.  if ( ng = empty ) then
28.      Report FAILURE;      end ← true ;
29.  else                                   /* Let x_j denote the variable on rhs(ng) */
30.      for k = j+1 to i−1  do  AgentView[k].value ← empty ; ;
31.      foreach ( nogood ∈ NogoodStore ) do
32.          if ( ¬Compatible(nogood, AgentView ) ∨  x_j ∈ nogood ) then
33.              remove(nogood, NogoodStore) ;
34.      AgentView.Consistent ← false;    v_i ← empty;
35.      sendMsg : ngd ⟨ ng⟩  to A_j ;

procedure ProcessNogood(msg)
36.  if ( Compatible(msg.nogood, AgentView ) ) then
37.      add(msg.nogood, NogoodStore) ;              /* according to the HPLV [19] */
38.      if ( rhs(msg.nogood).value = v_i ) then  v_i ← empty;  Assign() ;

procedure UpdateAgentView(CPA)
39.  AgentView ← CPA ;                              /* update values and tags */
40.  foreach ( ng ∈ NogoodStore ) do
41.      if ( ¬Compatible(ng, AgentView ) ) then  remove(ng,NogoodStore);

procedure Revise()
42.  foreach ( v ∈ D^0(x_i) ) do
43.      if ( v is ruled out by AgentView ) then
44.          store the best nogood for v;            /* according to the HPLV [19] */
```

Figure 2: Nogood-based AFC algorithm running by agent A_i (Part 2).

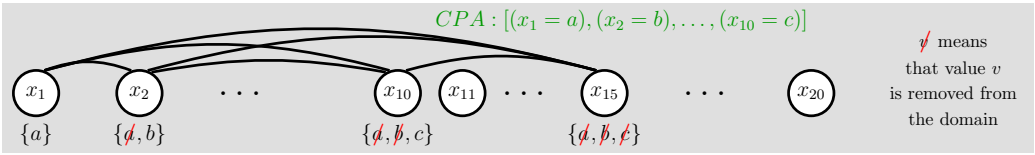
Whenever ***stp*** message is received, A_i marks *end* flag true to stop the main loop (line 9).

3.2 A simple example of the backtrack operation on AFC-like algorithms

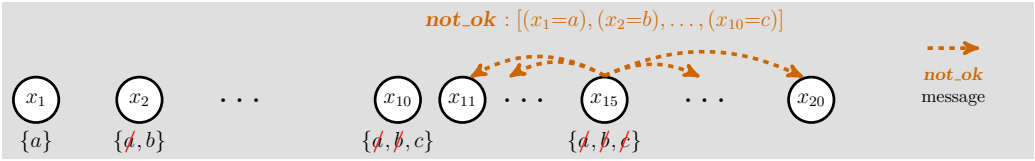
Figure 3 illustrates the backtrack operation on AFC, DBJ and AFC-ng when detecting a dead-end. Figure 3(a) shows a simple instance of a DisCSP containing 20 agents $\mathcal{X} = \{x_1, \ldots, x_{20}\}$. The domains of x_1, x_2, x_{10}, x_{15} are $D^0(x_1) = \{a, f\}$, $D^0(x_2) = \{a, b\}$, $D^0(x_{10}) = D(x_{15}) = \{a, b, c\}$, the others can be anything. The constraints are $x_1 \neq x_2$, $x_1 \neq x_{10}$, $x_1 \neq x_{15}$, $x_2 \neq x_{10}$, $x_2 \neq x_{15}$. Let us assume that the ordering on agents is the lexicographic ordering $[x_1, \ldots, x_{20}]$. Assume also that when trying to solve this instance the algorithms, i.e., AFC, DBJ and AFC-ng, fall in the same situation shown in Figure 3(b). Agent x_1 assigns value a from its domain and then x_2 removes value a from its domain and assigns value b (i.e., $x_2 = b$) when receiving the ***cpa*** from x_1. When receiving the CPA from x_2, agent x_{10} (resp. x_{15}) removes values a and b from $D(x_{10})$ (resp. $D(x_{15})$) because of constraints connecting x_{10} (resp. x_{15}) to x_1 and x_2. Assume that agents x_3 to x_9 assign values successfully. When agent x_{10} receives the CPA from x_9, it assigns the last value in $D(x_{10})$, i.e., $x_{10} = c$. Agent x_{10} sends the CPA to x_{11} and copies to the lower neighbors



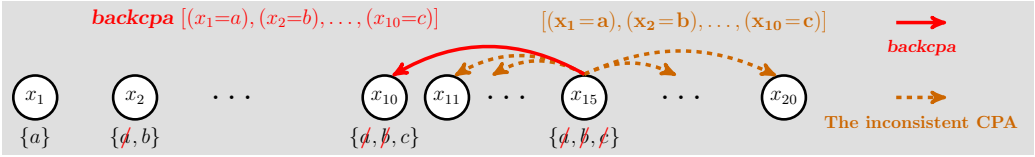
(a) A simple example of a DisCSP containing 20 agents



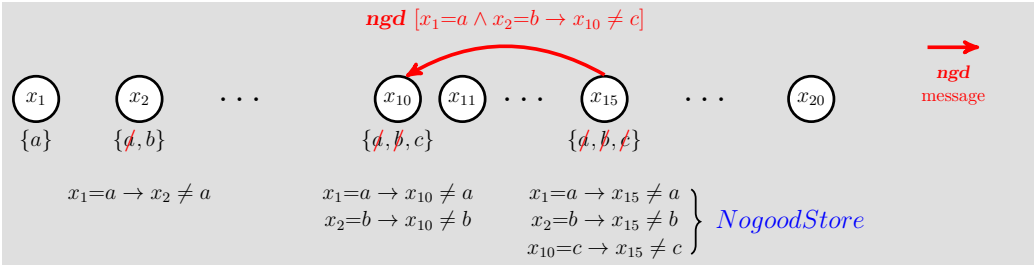
(b) The dead-end occurs on the domain of x_{15} after receiving the ***cpa*** $[(x_1 = a), (x_2 = b), \ldots, (x_{10} = c)]$



(c) In AFC, agent x_{15} initiates the backtrack operation by sending ***not_ok*** to unassigned agents



(d) In DBJ, agent x_{15} initiates the backtrack operation by sending the inconsistent CPA to unassigned agents and a ***backcpa*** to agent x_{10}



(e) In AFC-ng, agent x_{15} backtracks by sending a ***ngd*** to agent x_{10}

Figure 3: The backtrack operation on AFC, DBJ and AFC-ng on a simple example.

(including x_{15}). When receiving this copy of the CPA, x_{15} removes the last value from its domain generating a dead-end Figure 3(b).

In front of this situation of dead-end, AFC, DBJ and AFC-ng behave differently. In AFC (Figure 3(c)), agent x_{15} sends ***not_ok*** messages to unassigned agents (i.e., $[x_{11}, \ldots, x_{20}]$) informing them that the CPA $[x_1 = a, x_2 = b, \ldots, x_{10} = c]$ is inconsistent. Only the

agent who will receive the CPA from its predecessor when holding this ***not_ok*** (that is, one among $x_{11}, .., x_{14}$) will send the backtrack to x_{10}. In DBJ (Figure 3(d)), agent x_{15} backtracks directly to x_{10} and informs unassigned agents (i.e., $[x_{11}, \ldots, x_{20}]$) that the CPA $[x_1 = a, x_2 = b, \ldots, x_{10} = c]$ is inconsistent. In AFC-ng (Figure 3(e)), when agent x_{15} produces an empty domain after receiving the copy of the CPA from x_{10}, it resolves the nogoods from its NogoodStore (i.e., $[x_1 = a \rightarrow x_{15} \neq a]$, $[x_2 = b \rightarrow x_{15} \neq b]$ and $[x_{10} = c \rightarrow x_{15} \neq c]$). The resolved nogood $[x_1 = a \wedge x_2 = b \rightarrow x_{10} \neq c]$ is sent to agent x_{10} in a ***ngd*** message. In AFC-ng, we do not inform unassigned agents about the inconsistency of the CPA.

We are now in a situation where in all three algorithms AFC, DBJ and AFC-ng, x_{10} has received a backtrack message. After receiving the backtrack, x_{10} removes the last value, i.e., c, from $D(x_{10})$ and has to backtrack. In AFC and DBJ, x_{10} backtracks to x_9. We see that the backjump to x_{10} is followed by a backtrack step, as done by BJ in the centralized case, because BJ does not remember who were the other culprits of the initial backjump [16]. In AFC-ng, when x_{10} receives the backtrack from x_{15}, it removes value c and stores the received nogood as justification of its removal (i.e., $[x_1 = a \wedge x_2 = b \rightarrow x_{10} \neq c]$). After removing this last value, x_{10} resolves its nogoods generating a new nogood $[x_1 = a \rightarrow x_2 \neq b]$. Thus, x_{10} backtracks to x_2. We see that a new backjump follows the one to x_{10}. AFC-ng mimics the Conflict-directed BackJumping technique of the centralized case (CBJ) [32], which always jumps to the causes of the conflicts.

4 Asynchronous Forward Checking Tree

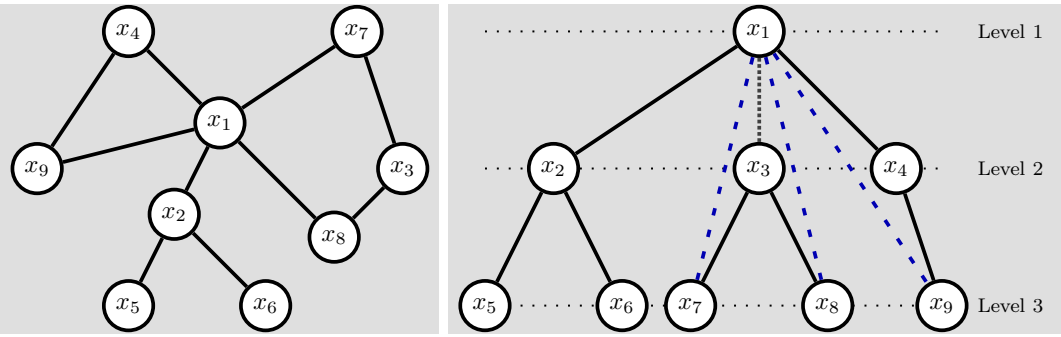
In this section, we show how to extend our AFC-ng algorithm to the *Asynchronous Forward Checking Tree (AFC-tree)* algorithm using a pseudo-tree arrangement of the constraint graph. To achieve this goal, agents are ordered a priori in a pseudo-tree such that agents in different branches of the tree do not share any constraint. AFC-tree does not address the process of ordering the agents in a pseudo-tree arrangement. Therefore, the construction of the pseudo-tree is done in a preprocessing step. Now, it is known from centralized CSPs that the performance of the search procedures tightly depends on the variable ordering. Thus, the task of constructing the pseudo-tree is important for a search algorithm like AFC-tree.

4.1 Pseudo-tree ordering

Any binary DisCSP can be represented by a *constraint graph* $G = (X_G, E_G)$, whose vertexes represent the variables and edges represent the constraints. Therefore, $X_G = \mathcal{X}$ and for each constraint $c_{ij} \in \mathcal{C}$ connecting two variables x_i and x_j there exists an edge $\{x_i, x_j\} \in E_G$ linking vertexes x_i and x_j.

The concept of *pseudo-tree* arrangement of a constraint graph has been introduced first by Freuder and Quinn in [15]. The purpose of this arrangement is to perform search in parallel on independent branches of the pseudo-tree in order to improve search in centralized constraint satisfaction problems.

Definition 6. *A* ***pseudo-tree*** *arrangement $T = (X_T, E_T)$ of a graph $G = (X_G, E_G)$ is a rooted tree with the same set of vertexes as G ($X_G = X_T$) such that vertexes in different branches of T do not share any edge in G.*



(a) a constraint graph G.

(b) a pseudo-tree arrangement T.

Figure 4: Example of a pseudo-tree arrangement of a constraint graph.

Figure 4(a) shows an example of a constraint graph G of a problem involving 9 variables $\mathcal{X} = X_G = \{x_1, \ldots, x_9\}$ and 10 constraints $\mathcal{C} = \{c_{12}, c_{14}, c_{17}, c_{18}, c_{19}, c_{25}, c_{26}, c_{37}, c_{38}, c_{49}\}$. An example of a pseudo-tree arrangement T of this constraint graph is illustrated in Figure 4(b). Notice that G and T have the same vertexes ($X_G = X_T$). However, a new (dotted) edge ($\{x_1, x_3\}$) linking x_1 to x_3 is added to T where $\{x_1, x_3\} \notin E_G$. Moreover, edges $\{x_1, x_7\}$, $\{x_1, x_8\}$ and $\{x_1, x_9\}$ belonging to the constraint graph G are not part of T. They are represented in T by dashed edges to show that constrained variables must be located in the same branch of T even if there is not an edge linking them.

From a pseudo-tree arrangement of the constraint graph we can define:

- A *branch* of the pseudo-tree is a path from the root to some leaf (e.g., $\{x_1, x_4, x_9\}$).
- A leaf is a vertex that has no child (e.g., x_9).
- The *children* of a vertex are its descendants connected to it through tree edges (e.g., `children`$(x_1) = \{x_2, x_3, x_4\}$).
- The *descendants* of a vertex x_i are vertexes belonging to the subtree rooted at x_i (e.g., `descendants`$(x_2) = \{x_5, x_6\}$ and `descendants`$(x_1) = \{\mathcal{X} \setminus x_1\}$).
- The *linked descendants* of a vertex are its descendants constrained with it together with its children, (e.g., `linkedDescendants`$(x_1) = \{x_2, x_3, x_4, x_7, x_8, x_9\}$).
- The *parent* of a vertex is the ancestor connected to it through a tree edge (e.g., `parent`$(x_9) = \{x_4\}$, `parent`$(x_3) = \{x_1\}$).
- A vertex x_i is an *ancestor* of a vertex x_j if x_i is the parent of x_j or an ancestor of the parent of x_j.
- The *ancestors* of a vertex x_i is the set of agents forming the path from the root to x_i's parent (e.g., `ancestors`$(x_8) = \{x_1, x_3\}$).

The construction of the pseudo-tree can be processed by a centralized procedure. First, a *system agent* must be elected to gather information about the constraint graph. Such system agent can be chosen using a leader election algorithm like that presented in [1]. Once, all

information about the constraint graph is gathered by the system agent, it can perform a centralized algorithm to build the pseudo-tree ordering. A decentralized modification of the procedure for building the pseudo-tree was introduced by Chechetka and Sycara in [8]. This algorithm allows the distributed construction of pseudo-trees without needing to deliver any global information about the whole problem to a single agent.

Whatever the method (centralized or distributed) for building the pseudo-tree, the obtained pseudo-tree may require the addition of some edges not belonging to the original constraint graph. In the example presented in Figure 4(b), a new edge linking x_1 to x_3 is added to the resulting pseudo-tree T. The structure of the pseudo-tree will be used for communication between agents. Thus, the added link between x_1 and x_3 will be used to exchange messages between them. However, in some distributed applications, the communication might be restricted to the neighboring agents (i.e., a message can be passed only locally between agents that share a constraint). The solution in such applications is to use a *depth-first search tree (DFS-tree)*. DFS-trees are special cases of pseudo-trees where all edges belong to the original graph.

We present in Figure 5 a simple distributed algorithm for the distributed construction of the DFS-tree named `DistributedDFS` algorithm. The `DistributedDFS` is similar to the algorithm proposed by Cheung in [10]. The `DistributedDFS` algorithm is a distribution of a DFS traversal of the constraint graph. Each agent maintains a set $Visited$ where it stores its neighbors which are already visited (line 2). The first step is to design the root

```
procedure DistributedDFS()
01. Select the root via a leader election algorithm ;
02. Visited ← ∅;  end ← false ;
03. if ( x_i is the elected root ) then  CheckNeighbourhood() ;
04. while ( ¬end ) do
05.     msg ← getMsg();
06.     Visited ← Visited ∪ (Γ(x_i) ∩ msg.VisitedAgents) ;
07.     if ( msg.Sender ∈ children(x_i) ) then
08.         descendants(x_i)← descendants(x_i) ∪ msg.VisitedAgents ;
09.     else
10.         parent(x_i)← msg.Sender ;
11.         ancestors(x_i)← msg.VisitedAgents ;
12.     CheckNeighbourhood();

procedure CheckNeighbourhood()
13. if ( Γ(x_i) = Visited ) then
14.     sendMsg : token ⟨ descendants(x_i) ∪ {x_i} ⟩  to parent(x_i) ;
15.     end ← true ;
16. else
17.     select x_j in Γ(x_i) \ Visited ;
18.     children(x_i) ← children(x_i) ∪ x_j ;
19.     sendMsg : token ⟨ ancestors(x_i) ∪ {x_i} ⟩  to A_j ;
```

Figure 5: The distributed depth-first search construction algorithm.

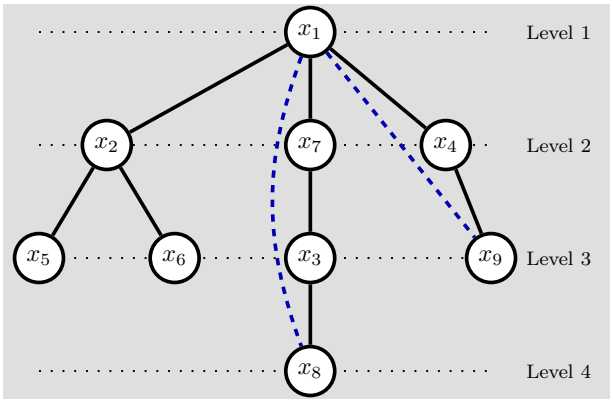


Figure 6: A DFS-tree arrangement of the constraint graph in Figure 4(a).

agent using a leader election algorithm (line 1). An example of leader election algorithm was presented by Abu-Amara in [1]. Once the root is designed, it can start the distributed construction of the DFS-tree (procedure `CheckNeighbourhood` call, line 3). The designed root initiates the propagation of a ***token***, which is a unique message that will be circulated on the network until "visiting" all the agents of the problem. The ***token*** massage contains a set of already visited agents, called $msg.VisitedAgents$.

When an agent x_i receives the ***token***, it marks all its neighbors included in the received message as visited (line 6). Next, x_i checks if the token is sent back by a child. If it is the case, x_i sets all agents belonging to the subtree rooted at message sender (i.e., its child) as its descendants (lines 7-8). Otherwise, the ***token*** is received for the first time from the parent of x_i. Thus, x_i marks the sender as its parent (line 10) and all agents contained in the token (i.e., the sender and its ancestors) as its ancestors (line 11). Afterwards, x_i calls the procedure `CheckNeighbourhood` to check if it has to pass on the ***token*** to an unvisited neighbor or to return back the ***token*** to its parent if all its neighbors are already visited.

The procedure `CheckNeighbourhood` checks if all neighbors are already visited (line 13). If it is the case, the agent x_i sends back the ***token*** to its parent (line 14). The ***token*** contains the set $VisitedAgents$ composed by x_i and its descendants. Until this point the agent x_i knows all its ancestors, its children and its descendants. Thus, the agent x_i terminates the execution of `DistributedDFS` (line 15). Otherwise, agent x_i chooses one of its neighbors (x_j) not yet visited and designs it as a child (lines 17-18). Afterwards, x_i passes on to x_j the ***token*** where it puts the ancestors of the child x_j (i.e., `ancestors(`x_i`)` $\cup \{x_i\}$) (line 19).

Consider for example the constraint graph G presented in Figure 4(a). Figure 6 shows an example of a DFS-tree arrangement of the constraint graph G obtained by performing distributively the `DistributedDFS` algorithm. The `DistributedDFS` algorithm can be performed as follows. First, let x_1 be the elected root of the DFS-tree (i.e., the leader election algorithm elects the most connected agent). The root x_1 initiates the DFS-tree construction by calling procedure `CheckNeighbourhood` (line 3). Then, x_1 selects from its unvisited neighbors x_2 to be its child (lines 17-18). Next, x_1 passes on the ***token*** to x_2 where it put itself to be the ancestor of the receiver (x_2) (line 19). After receiving the ***token***, x_2 updates the set of its visited neighbors (line 6) by marking x_1 (the only neighbor included in the ***token***) visited. Afterwards, x_2 sets x_1 to be its parent and puts $\{x_1\}$ to be its set

of ancestors (lines 10-11). Next, x_2 calls procedure `CheckNeighbourhood` (line 12). Until this point, x_2 has one visited neighbor (x_1) and two unvisited neighbors (x_5 and x_6). For instance, let x_2 chooses x_5 to be its child. Thus, x_2 sends the ***token*** to x_5 where it sets the set $msg.VisitedAgents$ to $\{x_1, x_2\}$. After receiving the ***token***, x_5 marks its single neighbor x_2 as visited (line 6), sets x_2 to be its parent (line 10), sets $\{x_1, x_2\}$ to be its ancestors ans sends the ***token*** back to x_2 where it puts itself. After receiving back the ***token*** from x_5, x_2 adds x_5 to its descendants and selects the last unvisited neighbor (x_6) to be its child and passes the ***token*** to x_6. In a similar way, x_6 returns back the ***token*** to x_2. Then, x_2 sends back the ***token*** to its parent x_1 since all its neighbors have been visited. The ***token*** contains the descendants of x_1 on the subtree rooted at x_2 (i.e., $\{x_2, x_5, x_6\}$). After receiving the ***token*** back from x_2, x_1 will select an agent from its unvisited neighbors $\{x_4, x_7, x_8, x_9\}$. Hence, the subtree rooted at x_2 where each agent knows its ancestors and its descendants is build without delivering any global information. The other subtrees respectively rooted at x_7 and x_4 are built in a similar manner. Thus, we obtain the DFS-tree shown in Figure 6.

4.2 The AFC-tree algorithm

The AFC-tree algorithm is based on AFC-ng performed on a pseudo-tree ordering of the constraint graph (built in a preprocessing step). Agents are prioritized according to the pseudo-tree ordering in which each agent has a single parent and various children. Using this priority ordering, AFC-tree performs multiple AFC-ng processes on the paths from the root to the leaves. The root initiates the search by generating a CPA, assigning its value on it, and sending CPA messages to its linked descendants. Among all agents that receive the CPA, children perform AFC-ng on the sub-problem restricted to its ancestors (agents that are assigned in the CPA) and the set of its descendants. Therefore, instead of giving the privilege of assigning to only one agent, agents who are in disjoint subtrees may assign their variables simultaneously. AFC-tree thus exploits the potential speed-up of a parallel exploration in the processing of distributed problems. The degree of asynchronism is enhanced.

An execution of AFC-tree on a simple DisCSP problem is shown in Figure 7. At time t_1, the root x_1 sends copies of the CPA on ***cpa*** messages to its linked descendants. Children x_2, x_3 and x_4 assign their values simultaneously in the received CPAs and then perform concurrently the AFC-tree algorithm. Agents x_7, x_8 and x_9 only perform a forward checking. At time t_2, x_9 finds an empty domain and sends a ***ngd*** message to x_1. At the same time, other CPAs propagate down through the other paths. For instance, a CPA has propagated down from x_3 to x_7 and x_8. x_7 detects an empty domain and sends a nogood to x_3 attached on a ***ngd*** message. For the CPA that propagates on the path (x_1, x_2, x_6), x_6 successfully assigned its value and initiated a solution detection. The same thing is going to happen on the path (x_1, x_2, x_5) when x_5 (not yet instantiated) will receive the CPA from its parent x_2. When x_1 receives the ***ngd*** message from x_9, it initiates a new search process by sending a new copy of the CPA which will dominate all other CPAs where x_1 is assigned its old value. This new CPA generated by x_1 can then take advantage from efforts done by the obsolete CPAs. Consider for instance the subtree rooted at x_2. If the value of x_2 is consistent with the value of x_1 on the new CPA, all nogoods stored on the subtree rooted at x_2 are still valid and a solution is reached on the subtree without any nogood generation.

In AFC-ng, a solution is reached when the last agent in the agent ordering receives the CPA and succeeds in assigning its variable. In AFC-tree, the situation is different because

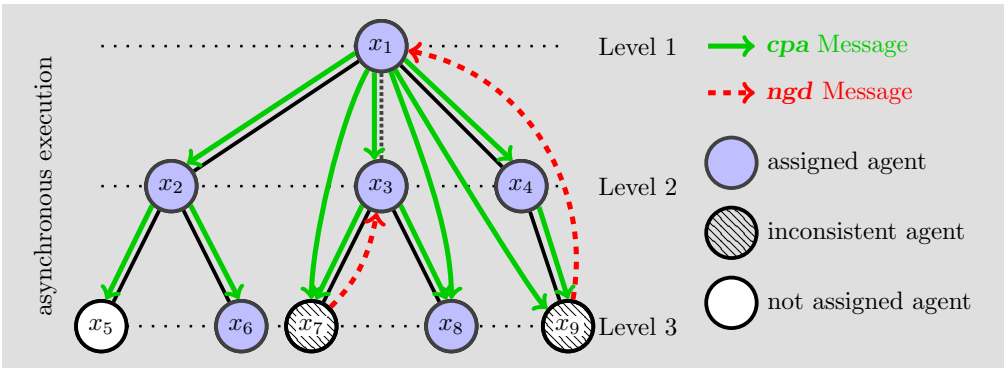


Figure 7: An example of the AFC-tree execution.

a CPA can reach a leaf agent without being complete. When all agents are assigned and no constraint is violated, this state is a global solution and the network has reached quiescence, meaning that no message is traveling through it. Such a state can be detected using specialized snapshot algorithms [7], but AFC-tree uses a different mechanism that allows to detect solutions before quiescence. AFC-tree uses an additional type of messages called ***accept*** that informs parents of the acceptance of their CPA. Termination can be inferred earlier and the number of messages required for termination detection can be reduced. A similar technique of solution detection was used in the AAS algorithm [34].

The mechanism of solution detection is as follows: whenever a leaf node succeeds in assigning its value, it sends an ***accept*** message to its parent. This message contains the CPA that was received from the parent incremented by the value-assignment of the leaf node. When a non-leaf agent A_i receives ***accept*** messages from all its children that are all compatible with each other, all compatible with A_i's AgentView and with A_i's value, A_i builds an ***accept*** message being the conjunction of all received ***accept*** messages plus A_i's value-assignment. If A_i is the root, a solution is found: A_i reports a solution and the process stops. Otherwise, A_i sends the built ***accept*** message to its parent.

Description of the algorithm

We present in Figure 8 only the procedures that are new or different from those of AFC-ng in Figures 1 and 2. In `InitAgentView`, the AgentView of A_i is initialized to the set `ancestors(`A_i`)` and t_j is set to 0 for each agent (x_j) in `ancestors(`A_i`)` (line 11). The new data structure storing the received ***accept*** messages is initialized to the empty set (line 12). In `SendCPA(CPA)`, instead of sending copies of the CPA to all neighbors not yet instantiated on it, A_i sends copies of the CPA to its linked descendants (`linkedDescendants(`A_i`)`, line 14). When the set `linkedDescendants(`A_i`)` is empty (i.e., A_i is a leaf), A_i calls the procedure `SolutionDetection` to build and send an ***accept*** message. In `CheckAssign(`*sender*`)`, A_i assigns its value if the CPA was received from its parent (line 17) (i.e., if *sender* is the parent of A_i).

In `ProcessAccept(`*msg*`)`, when A_i receives an ***accept*** message from its *child* for the first time, or the CPA contained in the received ***accept*** message is stronger than that received before, A_i stores the content of this message (lines 18-19) and calls the `SolutionDetection` procedure (line 20).

```
procedure AFC-tree()
01. InitAgentView();
02. end ← false; AgentView.Consistent ← true;
03. if ( A_i = root ) then Assign();
04. while ( ¬end ) do
05.     msg ← getMsg();
06.     switch ( msg.type ) do
07.         cpa     : ProcessCPA(msg);
08.         ngd     : ProcessNogood(msg);
09.         stp     : end ← true;
10.         accept  : ProcessAccept(msg);

procedure InitAgentView()
11. foreach ( A_j ∈ ancestors(A_i) ) do AgentView[j] ← {(x_j, empty, 0)} ;
12. foreach ( child ∈ children(A_i) ) do Accept[child] ← ∅ ;

procedure SendCPA(CPA)
13. if ( children(A_i) ≠ ∅ ) then
14.     foreach ( desc ∈ linkedDescendants(A_i) ) do
15.         sendMsg: cpa ⟨ CPA ⟩ to desc
16. else SolutionDetection() ;

procedure CheckAssign(sender)
17. if ( parent(A_i) = sender ) then Assign() ;

procedure ProcessAccept(msg)
18. if ( msg.CPA stronger than Accept[msg.Sender] ) then
19.     Accept[msg.Sender] ← msg.CPA ;
20.     SolutionDetection() ;

procedure SolutionDetection()
21. if ( children(A_i) = ∅ ) then
22.     sendMsg: accept ⟨ AgentView ∪ {(x_i, v_i, t_i)}⟩ to parent(A_i) ;
23. else
24.     PA ← BuildAccept() ;
25.     if ( PA ≠ ∅ ) then
26.         if ( A_i = root ) then Report SOLUTION; end ← true ;
27.         else sendMsg: accept ⟨ PA⟩ to parent(A_i) ;

function BuildAccept()
28. PA ← AgentView ∪ {(x_i, v_i, t_i)} ;
29. foreach ( child ∈ children(x_i) ) do
30.     if ( Accept[child] = ∅ ∨ ¬Compatible(PA, Accept[child]) ) then return ∅ ;
31.     else PA ← PA ∪ Accept[child] ;
32. return PA;
```

Figure 8: New lines/procedures of AFC-tree with respect to AFC-ng.

In `SolutionDetection`, if A_i is a leaf (i.e., `children`(A_i) is empty, line 21), it sends an ***accept*** message to its parent. The ***accept*** message sent by A_i contains its AgentView incremented by its own assignment (lines 21-22). If A_i is not a leaf, it calls function `BuildAccept` to build an accept partial solution PA (line 24). If the returned partial solution PA is not empty and A_i is the root, PA is a solution of the problem. Then, A_i reports a solution and sets the *end* flag to true to stop the search (line 26). Otherwise, A_i sends an ***accept*** message containing PA to its parent (line 27).

In `BuildAccept`, if an accept partial solution is reached. A_i generates a partial solution PA incrementing its AgentView with its assignment (line 28). Next, A_i loops over the set of ***accept*** messages received from its children. If at least one *child* has never sent an ***accept*** message or the ***accept*** message is incompatible with PA, then the partial solution has not yet been reached and the function returns empty (line 30). Otherwise, the partial solution PA is incremented by the ***accept*** message of *child* (line 31). Finally, the accept partial solution is returned (line 32).

5 Correctness Proofs

Theorem 1. *The spatial complexity of AFC-ng (resp. AFC-tree) is polynomially bounded by $O(nd)$ per agent.*

Proof. In AFC-ng, the size of nogoods is bounded by n, the total number of variables. In AFC-tree, the size of nogoods is bounded by h ($h \leq n$), the height of the pseudo-tree. Now, on each agent, AFC-ng (resp. AFC-tree) only stores one nogood per removed value. Thus, the space complexity of AFC-ng is in $O(nd)$ on each agent. AFC-tree also stores its set of descendants and ancestors, which is bounded by n on each agent. Therefore, AFC-tree has a space complexity in $O(hd + n)$. □

Lemma 1. *AFC-ng is guaranteed to terminate.*

Proof. We prove by induction on the agent ordering that there will be a finite number of new generated CPAs (at most d^n, where d is the size of the initial domain and n the number of variables.), and that agents can never fall into an infinite loop for a given CPA. The base case for induction ($i = 1$) is obvious. The only messages that x_1 can receive are ***ngd*** messages. All nogoods contained in these ***ngd*** messages have an empty lhs. Hence, values on their rhs are removed once and for all from the domain of x_1. Now, x_1 only generates a new CPA when it receives a nogood ruling out its current value. Thus, the maximal number of CPAs that x_1 can generate equals the size of its initial domain (d). Suppose now that the number of CPAs that agents $x_1, \ldots, x_{i-1}$ can generate is finite (and bounded by d^{i-1}). Given such a CPA on $[x_1, \ldots, x_{i-1}]$, x_i generates new CPAs (line 13, Figure 1) only when it changes its assignment after receiving a nogood ruling out its current value v_i. Given the fact that any received nogood can include, in its lhs, only the assignments of higher priority agents ($[x_1, \ldots, x_{i-1}]$), this nogood will remain valid as long as the CPA on $[x_1, \ldots, x_{i-1}]$ does not change. Thus, x_i cannot regenerate a new CPA containing v_i without changing assignments on higher priority agents ($[x_1, \ldots, x_{i-1}]$). Since there are a finite number of values on the domain of variable x_i, there will be a finite number of new CPAs generated by x_i (d^i). Therefore, by induction we have that there will be a finite number of new CPAs (d^n) generated by AFC-ng.

Let *cpa* be the strongest CPA generated in the network and A_i be the agent that generated *cpa*. After a finite amount of time, all unassigned agents on *cpa* ($[x_{i+1}, \ldots, x_n]$) will receive *cpa* and thus will discard all other CPAs. Two cases occur. First case, at least one agent detects a dead-end and thus backtracks to an agent A_j included in *cpa* (i.e., $j \leq i$) forcing it to change its current value on *cpa* and to generate a new stronger CPA. Second case (no agent detects dead-end), if $i < n$, A_{i+1} generates a new stronger CPA by adding its assignment to *cpa*, else ($i = n$), a solution is found. As a result, agents can never fall into an infinite loop for a given CPA and AFC-ng is thus guaranteed to terminate. □

Lemma 2. *AFC-ng cannot infer inconsistency if a solution exists.*

Proof. Whenever a stronger CPA or a ***ngd*** message is received, AFC-ng agents update their NogoodStore. Hence, for every CPA that may potentially lead to a solution, agents only store valid nogoods. In addition, every nogood resulting from a CPA is redundant with regard to the DisCSP to solve. Since all additional nogoods are generated by logical inference when a domain wipe-out occurs, the empty nogood cannot be inferred if the network is satisfiable. This mean that AFC-ng is able to produce all solutions. □

Theorem 2. *AFC-ng is correct.*

Proof. The argument for soundness is close to the one given in [26, 28]. The fact that agents only forward consistent partial solution on the CPA messages at only one place in procedure `Assign` (line 14, Figure 1), implies that the agents receive only consistent assignments. A solution is found by the last agent only in procedure `SendCPA(CPA)` at line 17 of Figure 1. At this point, all agents have assigned their variables, and their assignments are consistent. Thus the AFC-ng algorithm is sound. Completeness comes from the fact that AFC-ng is able to terminate and does not report inconsistency if a solution exists (Lemmas 1 and 2). □

Theorem 3. *AFC-tree algorithm is correct.*

Proof. AFC-tree agents only forward consistent partial assignments (CPAs). Hence, leaf agents receive only consistent CPAs. Thus, leaf agents only send ***accept*** messages holding consistent assignments to their parent. A parent A_i builds an ***accept*** message only when the ***accept*** messages received from all its children are compatible with each other and are all compatible with its AgentView and its own value. As a result, the ***accept*** message A_i sends to its own parent contains a consistent partial solution. The root reports a solution and stops the search only when it can build itself such an ***accept*** message. Therefore, the solution is correct and AFC-tree is sound.

From Lemma 1 we deduce that the AFC-tree agent of highest priority cannot fall into an infinite loop. By induction on the level of the pseudo-tree no agent can fall in such a loop, which ensures the termination of AFC-tree. AFC-tree performs multiple AFC-ng processes on the paths of the pseudo-tree from the root to the leaves. Thus, from Lemma 2, AFC-tree inherits the property that an empty nogood cannot be inferred if the network is satisfiable. As AFC-tree terminates, this ensures its completeness. □

6 Experimental Evaluation

In this section we experimentally compare our algorithms, i.e., AFC-ng and AFC-tree, to other well known algorithms both with a static variable ordering behavior and a dynamic

variable ordering behavior. Algorithms are evaluated on three benchmarks: Uniform Binary Random DisCSPs, Distributed Sensor-Mobile and Distributed Meeting Scheduling.

All experiments were performed on the DisChoco 2.0 platform[2] [35], in which agents are simulated by Java threads that communicate only through message passing. We evaluate the performance of the algorithms by communication load [22] and computation effort. Communication load is measured by the total number of exchanged messages among agents during algorithm execution ($\#msg$), including those of termination detection (system messages). Computation effort is measured by the number of non-concurrent constraint checks ($\#ncccs$) [42]. $\#ncccs$ is the metric used in distributed constraint solving to simulate the computation time. In DisChoco 2.0, nogood checks are considered as constraint checks. Thus, the additional computational effort performed by nogood-based algorithms for handling nogoods is taken into account in the $\#ncccs$ measure.

Section 6.1 describes the three types of benchmarks, Section 6.2 presents the results on algorithms with a static variable ordering behavior, Section 6.3 presents the results on algorithms with a dynamic variable ordering behavior. Finally, Section 6.4 draws the conclusions of these experiments.

6.1 The three benchmarks

6.1.1 Uniform binary random DisCSPs

Uniform binary random DisCSPs are characterized by $\langle n, d, p_1, p_2\rangle$, where n is the number of agents/variables, d is the number of values in each of the domains, p_1 is the network connectivity defined as the ratio of existing binary constraints, and p_2 is the constraint tightness defined as the ratio of forbidden value pairs. We solved instances of two classes of constraint graphs: sparse graphs $\langle 20, 10, 0.2, p_2\rangle$ and dense graphs $\langle 20, 10, 0.7, p_2\rangle$. We varied the tightness from 0.1 to 0.9 by steps of 0.05. For each pair of fixed density and tightness (p_1, p_2) we generated 25 instances, solved 4 times each. We report average over the 100 runs.

6.1.2 Distributed sensor-mobile problems

The *Distributed Sensor-Mobile Problem* (SensorDisCSP) [2] is a benchmark based on a real distributed problem. It consists of n sensors that track m mobiles. Each mobile must be tracked by 3 sensors. Each sensor can track at most one mobile. A solution must satisfy visibility and compatibility constraints. The visibility constraint defines the set of sensors to which a mobile is visible. The compatibility constraint defines the compatibility among sensors.

We encode SensorDisCSP in DisCSP as follows. Each agent represents one mobile. There are three variables per agent, one for each sensor that we need to allocate to the corresponding mobile. The domain of each variable is the set of sensors that can detect the corresponding mobile. The intra-agent constraints between the variables of one agent (mobile) specify that the three sensors assigned to the mobile must be distinct and pairwise compatible. The inter-agent constraints between the variables of different agents specify that a given sensor can be selected by at most one agent. In our implementation of the DisCSP algorithms, this encoding is translated into an equivalent formulation where we have three virtual agents for each real agent. Each virtual agent handles a single variable

[2] http://www.lirmm.fr/coconut/dischoco/

but $\#msg$ does not take into account messages exchanged between virtual agents belonging to the same real agent.

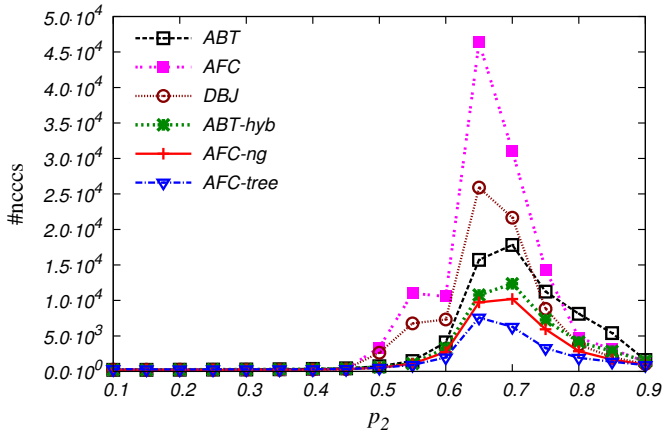
Problems are characterized by $\langle n, m, p_c, p_v \rangle$, where n is the number of sensors, m is the number of mobiles, each sensor can communicate with a fraction p_c of the sensors that are in its sensing range, and each mobile can be tracked by a fraction p_v of the sensors having the mobile in their sensing range. We solved instances for the class $\langle 25, 5, 0.4, p_v \rangle$, where we vary p_v from 0.1 to 0.9 by steps of 0.05. Again, for each pair (p_c, p_v) we generated 25 instances, solved 4 times each, and averaged over the 100 runs.

6.1.3 Distributed meeting scheduling problems

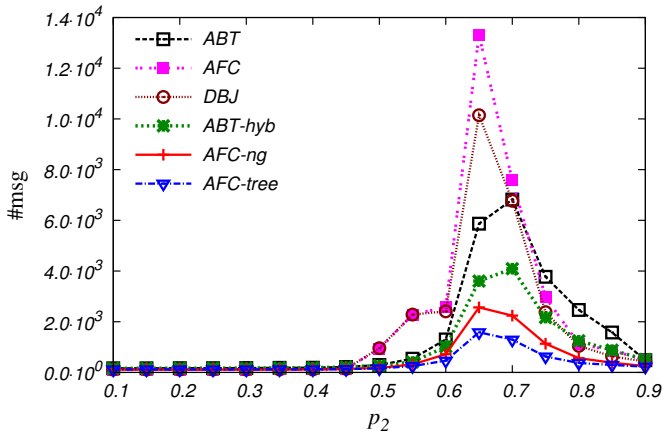
The *Distributed Meeting Scheduling Problem* (DMSP) is a truly distributed benchmark where agents may not desire to deliver their personal information to a centralized agent to solve the whole problem [36, 24]. The DMSP consists of a set of n agents having a personal private calendar and a set of m meetings each taking place in a specified location. Each agent knows the set of the k among m meetings he/she must attend. It is assumed that each agent knows the traveling time between the locations where his/her meetings will be held. The traveling time between two meetings m_i and m_j is denoted by $TravelingTime(m_i, m_j)$. Solving the problem consists in satisfying the following constraints: (i) all agents attending a meeting must agree on when it will occur, (ii) an agent cannot attend two meetings at same time, (iii) an agent must have enough time to travel from the location where he/she is to the location where the next meeting will be held.

We encode the DMSP in DisCSP as follows. Each DisCSP agent represents a real agent and contains k variables representing the k meetings to which the agent participates. These k meetings are selected randomly among the m meetings. The domain of each variable contains the $d \times h$ slots where a meeting can be scheduled. A slot is one hour long, and there are h slots per day and d days. There is an equality constraint for each pair of variables corresponding to the same meeting in different agents. This equality constraint means that all agents attending a meeting must schedule it at the same slot (constraint (i)). There is an *arrival-time* constraint between all variables/meetings belonging to the same agent. The arrival-time constraint between two variables m_i and m_j is $|m_i - m_j| - duration > TravelingTime(m_i, m_j)$, where $duration$ is the duration of every meeting. This arrival-time constraint allows us to express both constraints (ii) and (iii). We place meetings randomly on the nodes of a uniform grid of size $g \times g$ and the traveling time between two adjacent nodes is 1 hour. Thus, the traveling time between two meetings equals the Euclidean distance between nodes representing the locations where they will be held. For varying the tightness of the arrival-time constraint we vary the size of the grid on which meetings are placed.

Problems are characterized by $\langle n, m, k, d, h, g \rangle$, where n is the number of agents, m is the number meetings, k is the number of meetings/variables per agent, d is the number of days and h is the number of hours per day, and g is the grid size. The duration of each meeting is one hour. In our implementation of the DisCSP algorithms, this encoding is translated into an equivalent formulation where we have k (number of meetings per agent) virtual agents for each real agent. Each virtual agent handles a single variable but $\#msg$ does not take into account messages exchanged between virtual agents belonging to the same real agent. We solved instances for the class $\langle 20, 9, 3, 2, 10, g \rangle$ where we vary g from 2 to 22 by steps of 2. Again, for each g we generated 25 instances, solved 4 times each, and averaged over the 100 runs.



(a) $\#ncccs$ performed on sparse random DisCSPs



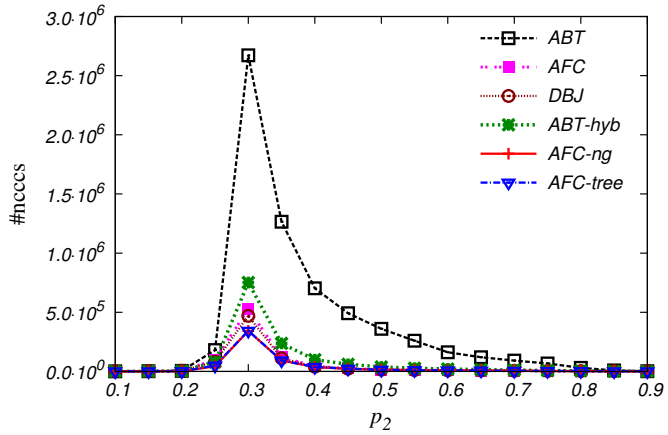
(b) $\#msg$ sent on sparse random DisCSPs

Figure 9: Total number of messages sent and $\#ncccs$ performed on sparse uniform binary random DisCSPs problems where $\langle n = 20, d = 10, p_1 = 0.2 \rangle$.

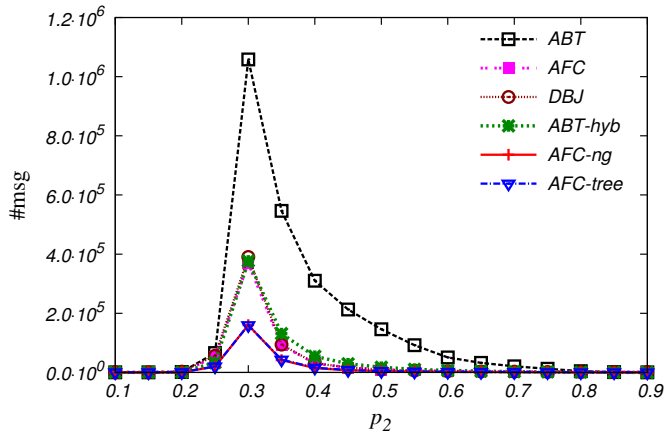
6.2 Static variable ordering

We first compare AFC-ng and AFC-tree to algorithms with a static variable ordering behavior. These algorithms are: ABT [40, 4], AFC [26], DBJ [28] and ABT-hyb [5]. All algorithms are tested on the same static agents ordering using the dom/deg heuristic [3] and the same nogood selection heuristic ($HPLV$) [19]. For ABT and ABT-hyb we implemented a solution detection mechanism derived from Silaghi's solution detection [33] and counters for tagging assignments.

Figure 9 presents the performance of ABT, AFC, DBJ, ABT-hyb, AFC-ng and AFC-tree running on the uniform binary random sparse instances ($p_1 = 0.2$). In terms of computational effort (Figure 9(a)), we observe that at the complexity peak, AFC is the less efficient algorithm. DBJ improves on AFC by a factor of 1.7. ABT-hyb and ABT are better than



(a) $\#ncccs$ performed on dense random DisCSPs

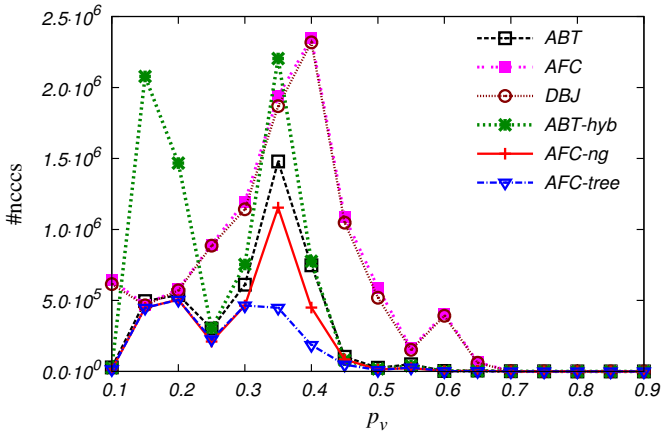


(b) $\#msg$ sent on dense random DisCSPs

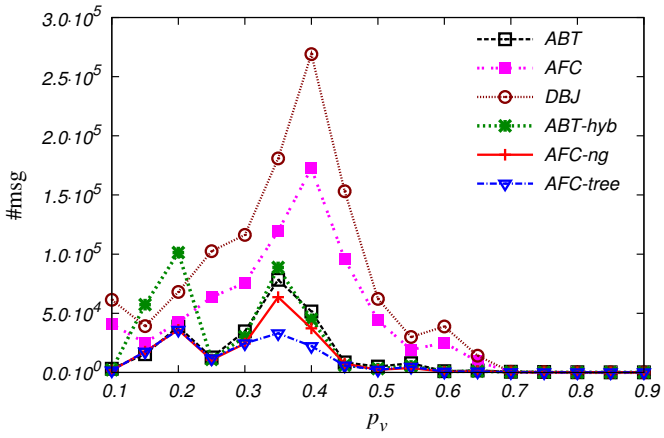
Figure 10: Total number of messages sent and $\#ncccs$ performed on dense uniform binary random DisCSPs problems where $\langle n = 20, d = 10, p_1 = 0.7 \rangle$.

DBJ. On the most difficult instances, AFC-ng improves the performance of standard AFC by a factor of 4.7, it outperforms ABT by a factor of 1.6, and is slightly better than ABT-hyb. AFC-tree takes advantage of the pseudo-tree arrangement to be the best of all algorithms. Concerning communication load (Figure 9(b)), AFC is again the worst algorithm and DBJ is the second worst. AFC-ng improves AFC by a factor of 5.2 and ABT-hyb by a factor of 1.5 whereas AFC-tree has the smallest communication load on all problems.

Figure 10 presents the results on the uniform binary random dense instances ($p_1 = 0.7$). When comparing the computational effort (Figure 10(a)), the results show that ABT dramatically deteriorates compared to other algorithms. AFC and DBJ perform almost the same $\#ncccs$ and they outperform ABT-hyb. AFC-ng and AFC-tree show a small improvement compared to AFC and DBJ. Regarding communication load (Figure 10(b)),



(a) $\#ncccs$ performed on SensorDisCSP

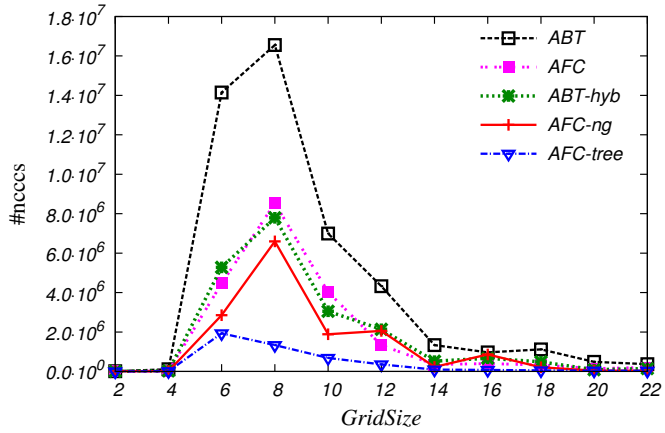


(b) $\#msg$ sent on SensorDisCSP

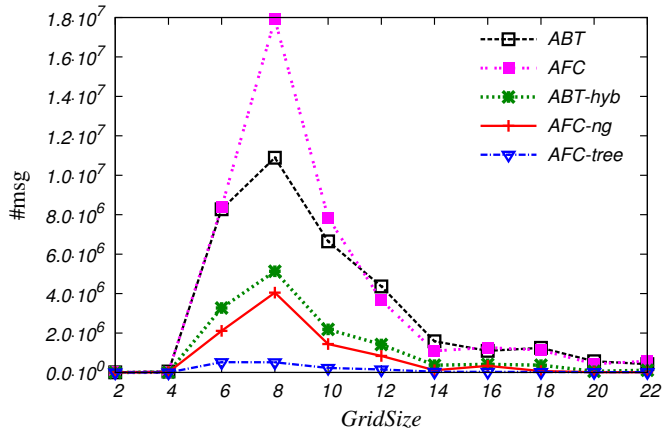
Figure 11: Total number of messages sent and $\#ncccs$ performed on instances of distributed sensor-mobile problems where $\langle n = 25, m = 5, p_c = 0.4\rangle$.

ABT is again significantly the worst. AFC, DBJ and ABT-hyb require almost the same number of messages. AFC-ng and AFC-tree outperform them by a factor of 2.5. On these dense graphs, AFC-tree behaves like AFC-ng because it does not benefit from the pseudo-tree arrangement, which in such graphs is a 'chain-like' pseudo-tree.

Figure 11 presents the results obtained on SensorDisCSP with $\langle n = 25, m = 5, p_c = 0.4\rangle$. When comparing the computational effort (Figure 11(a)), DBJ behaves like AFC and they are the less efficient algorithms. The performance of ABT-hyb is unstable (strong deterioration in the interval [0.15 0.20]). ABT is better than ABT-hyb but still, AFC-ng outperforms it. AFC-tree outperforms all the compared algorithms. On the most difficult instances, AFC-tree outperforms AFC and DBJ by a factor of 4.3, ABT by a factor of 3.3, and AFC-ng by a factor of 2.5. Concerning communication load (Figure 11(b)), DBJ is the



(a) $\#ncccs$ performed on DMSP

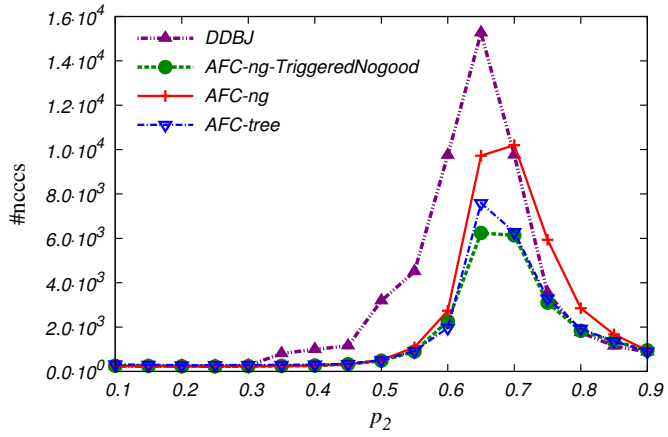


(b) $\#msg$ sent on DMSP

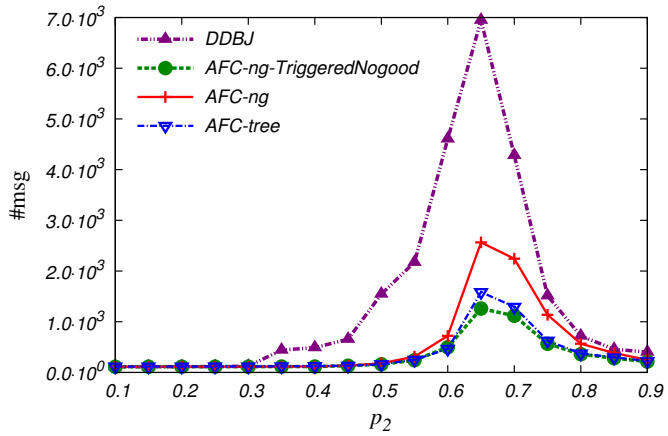
Figure 12: Total number of messages sent and $\#ncccs$ performed on meeting scheduling benchmarks where $\langle n = 20,\ m = 9,\ k = 3, d = 2, h = 10\rangle$.

worst algorithm and AFC is the second worst. ABT-hyb exchanges more messages than all other algorithms in the interval [0.15 0.20] and requires almost the same number of messages as ABT outside this interval. Between ABT and AFC-ng the difference is smaller than that on the computation effort. AFC-tree remains the best on all problems. On the most difficult instances, AFC-tree outperforms ABT and ABT-hyb by a factor of 2.5, and AFC-ng by a factor of 2.

Figure 12 presents the results obtained on the DMSP $\langle n = 20,\ m = 9,\ k = 3,\ d = 2,\ h = 10\rangle$. AFC-tree and AFC-ng continue to perform well. AFC-tree is significantly better that all other algorithms, both for computational effort (Figure 12(a)) and communication load (Figure 12(b)). When comparing the speed-up of algorithms (Figure 12(a)), ABT is the slowest algorithm to solve such problems. AFC outperforms ABT by a factor of 2 at the



(a) $\#ncccs$ performed on sparse random DisCSPs



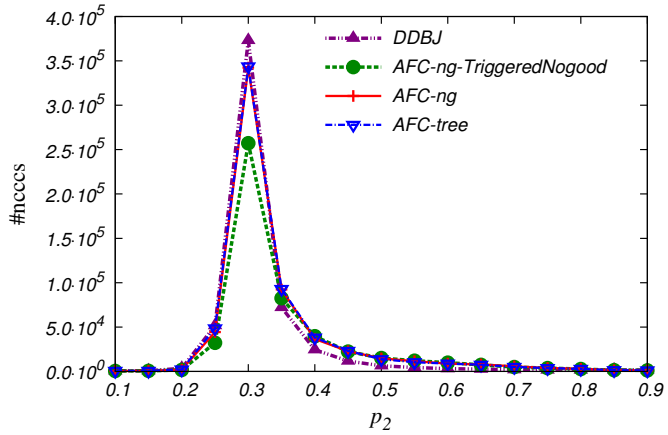
(b) $\#msg$ sent on sparse random DisCSPs

Figure 13: Total number of messages sent and $\#ncccs$ performed on sparse uniform binary random DisCSPs problems where $\langle n = 20, d = 10, p_1 = 0.2 \rangle$.

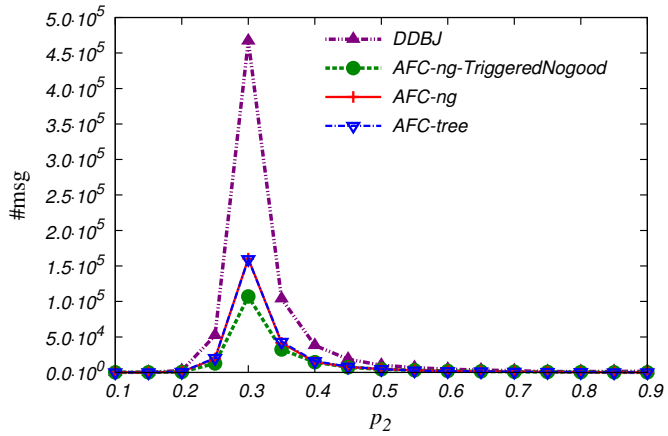
peak. However, ABT requires less messages than AFC. AFC-ng and ABT-hyb are close to each other, with a slight gain for AFC-ng.

6.3 Dynamic variable ordering

In synchronous algorithms agents perform assignments one by one in a sequential way. Implementing dynamic variable ordering heuristics in these algorithms is quite natural and not too difficult. More concretely, when an agent succeeds in extending the CPA by assigning its variable on it, it can choose the agent who will assign next its variable. In the following, we want to assess the performance of our static AFC-ng and AFC-tree algorithms when compared to an algorithm that uses dynamic variable ordering, as DDBJ [28]. DDBJ inte-



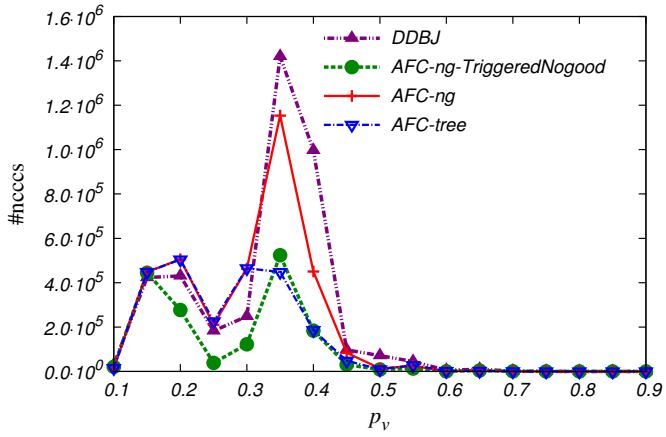
(a) *#ncccs* performed on sparse random DisCSPs



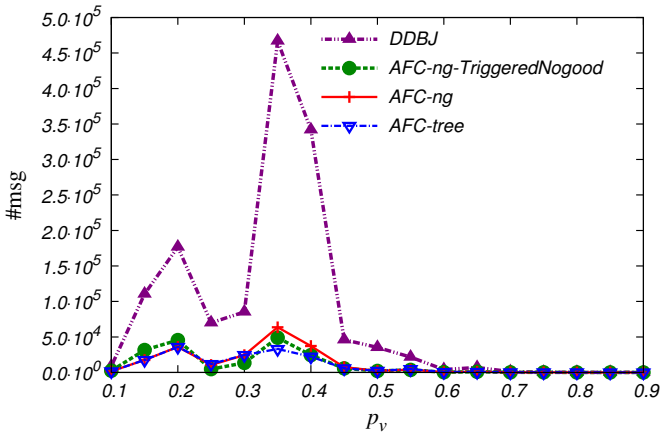
(b) *#msg* sent on sparse random DisCSPs

Figure 14: Total number of messages sent and *#ncccs* performed on dense uniform binary random DisCSPs problems where $\langle n = 20, d = 10, p_1 = 0.7 \rangle$.

grates dynamic value and variable ordering heuristics called the *possible conflict heuristics* on the DBJ. DDBJ uses additional messages to compute the dynamic ordering heuristics. Meisels and Zivan have shown empirically for random DisCSPs that DDBJ is better than AFC in *#ncccs* and that DDBJ is also better than the version of AFC augmented with dynamic variable ordering heuristics [26]. To make our comparison complete, we implemented a version of AFC-ng with the *nogood-triggered* dynamic variable ordering heuristic [43]. The *nogood-triggered* heuristic is inspired by Dynamic Backtracking [17] and was first implemented for ABT in [43] and adapted for AFC in [26]. This heuristic does not require any additional messages to be implemented in AFC-like algorithms. When an agent takes a new value due to a backtrack (***ngd*** message), it chooses the sender of the backtrack to be the next agent to assign its variable. It thus sends the CPA to that ***ngd*** sender, and tells it that it is its new successor.



(a) $\#ncccs$ performed on SensorDisCSP

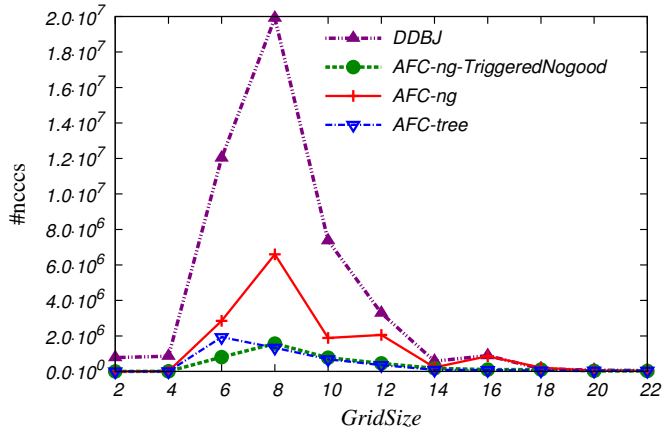


(b) $\#msg$ sent on SensorDisCSP

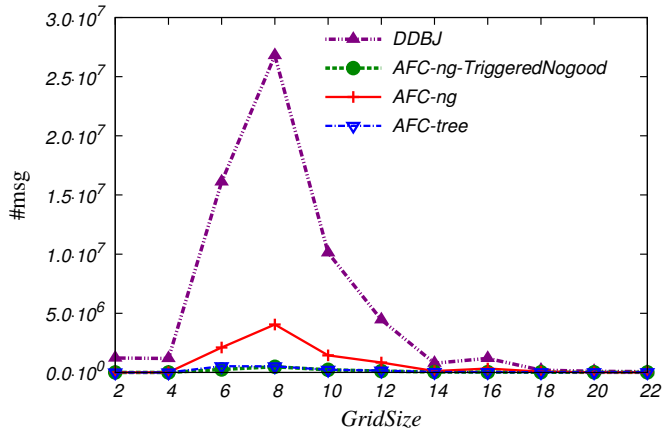
Figure 15: Total number of messages sent and $\#ncccs$ performed on instances of distributed sensor-mobile problems where $\langle n = 25, m = 5, p_c = 0.4\rangle$.

Figure 13 presents the performance of DDBJ, AFC-ng-TriggeredNogood, AFC-ng and AFC-tree running on the uniform binary random sparse instances where $\langle n = 20, d = 10, p_1 = 0.2\rangle$. When comparing the computational effort of algorithms (Figure 13(a)), we observe that at the complexity peak, DDBJ is the less efficient algorithm. It is better than AFC-ng only on instances to the right of the complexity peak (over-constrained). AFC-ng-TriggeredNogood improves the performance of static AFC-ng and also outperforms AFC-tree. Concerning communication load (Figure 13(b)), DDBJ dramatically deteriorates compared to the other algorithms. AFC-ng-TriggeredNogood improves AFC-ng by a factor of 2 and shows a slight gain compared to AFC-tree.

Figure 14 presents the results on the uniform binary random dense instances where $\langle n = 20, d = 10, p_1 = 0.7\rangle$. AFC-ng and AFC-tree have the same performance in dense



(a) $\#ncccs$ performed on DMSP



(b) $\#msg$ sent on DMSP

Figure 16: Total number of messages sent and $\#ncccs$ performed on meeting scheduling benchmarks where $\langle n = 20,\ m = 9,\ k = 3, d = 2, h = 10\rangle$.

graphs because of the chain-like pseudo-tree phenomenon seen in Section 6.2. In terms of computational effort (Figure 14(a)), AFC-ng and AFC-tree slightly outperform DDBJ at the complexity peak. AFC-ng-TriggeredNogood is the best. When comparing the communication load (Figure 14(b)), DDBJ is again significantly the worst algorithm. AFC-ng-TriggeredNogood requires less messages than AFC-ng and AFC-tree.

Figure 15 presents the results obtained on SensorDisCSP with $\langle n = 25,\ m = 5,\ p_c = 0.4\rangle$. Regarding the computational effort (Figure 15(a)), we observe that AFC-ng and AFC-tree are slightly dominated by DDBJ in the interval [0.20 0.30]. In this interval AFC-ng-TriggeredNogood is the fastest algorithm. At the complexity peak, AFC-ng outperforms DDBJ. AFC-ng-TriggeredNogood improves AFC-ng by a factor of 2.2 even though it is marginally dominated by AFC-tree. When comparing the communication load (Fig-

ure 15(b)), DDBJ is again the algorithm that requires the largest number of messages. It is widely dominated by the other algorithms. AFC-tree outperforms DDBJ by a factor of 14. AFC-ng, AFC-ng-TriggeredNogood and AFC-tree exchange almost the same number of messages with a slight gain for AFC-tree at the complexity peak.

Figure 16 presents the results obtained on DMSP where $\langle n = 20, m = 9, k = 3, d = 2, h = 10\rangle$. The results show that DDBJ dramatically deteriorates compared to other algorithms, both for computational effort (Figure 16(a)) and communication load (Figure 16(b)). On this class of meeting scheduling problems, AFC-ng-TriggeredNogood improves AFC-ng by a significant scale. AFC-ng-TriggeredNogood and AFC-tree require the same computational effort and communication load, except for $Gridsize = 6$, where AFC-ng-TriggeredNogood requires less $\#ncccs$ than AFC-tree.

6.4 Discussion

A first observation on these experiments is that AFC-ng is always better than AFC and DBJ, both in terms of computational effort ($\#ncccs$) and communication load ($\#msg$). A closer look at the type of exchanged messages shows that the backtrack operation in AFC and DBJ requires exchanging a lot of messages (containing the inconsistent CPA) (approximately 50% of the total number of messages sent by agents in AFC and 35% in DBJ). AFC and DBJ send these messages to unassigned agents to stop the current forward checking phase and wait for a newer CPA. Instead of stopping these forward checking phases, AFC-ng lets them alive and takes advantage of their former computations by keeping those of their nogoods that are compatible with the new arriving CPA. In addition, mimicking the CBJ mechanism allows AFC-ng to jump to the culprit agent. AFC-ng saves unnecessary search effort that may be done both by AFC and by DBJ. A second observation on these experiments is that AFC-tree is almost always better than or equivalent to AFC-ng both in terms of computational effort and communication load. When the graph is sparse, AFC-tree benefits from running separate search processes in disjoint problem subtrees. When the graph is dense, AFC-tree runs on a chain-like pseudo-tree and thus mimics AFC-ng. A third observation on these experiments is that DDBJ suffers from the extra-messages it exchanges to perform its dynamic value and variable ordering heuristic. It is usually dominated by AFC-ng, which uses a static variable ordering. A fourth observation on these experiments is that AFC-ng-TriggeredNogood is always better than AFC-ng. In addition, AFC-ng-TriggeredNogood is almost always better than or equivalent to AFC-tree both in terms of computational effort and communication load. This tends to show that on structured instances (where AFC-tree was better than AFC-ng), the dynamic reordering of variables allows AFC-ng-TriggeredNogood to adapt the order to the structure. A final observation is that ABT performs badly in dense graphs compared to algorithms that perform assignments on a sequential way.

7 Conclusion

In this paper, we have proposed two new complete algorithms. The first algorithm, Nogood-Based Asynchronous Forward Checking (AFC-ng), is an improvement on AFC. Besides its use of nogoods as justification of value removals, AFC-ng allows simultaneous backtracks going from different agents to different destinations. Thus, AFC-ng draws all the benefit it can from the asynchronism of the forward checking phase. The second algorithm, Asynchronous

Forward Checking Tree (AFC-tree), is based on AFC-ng and is performed on a pseudo-tree arrangement of the constraint graph. AFC-tree runs simultaneous AFC-ng processes on each branch of the pseudo-tree to exploit the parallelism inherent in the problem. Our experiments show that AFC-ng improves the AFC algorithm in terms of computational effort and number of exchanged messages. Experiments also show that AFC-tree is the most robust algorithm. It is particularly good when the problems are sparse because it takes advantage of the problem structure. We finally implemented AFC-ng-TriggeredNogood, a version of AFC-ng that implements the *nogood-triggered* dynamic variable ordering heuristic. AFC-ng-TriggeredNogood improves AFC-ng and is almost always better than or equivalent to AFC-tree both in terms of computational effort and communication load. It also outperforms DDBJ, which was a former synchronous algorithm with dynamic variable ordering.

References

[1] Hosame H. Abu-Amara. Fault-Tolerant Distributed Algorithm for Election in Complete Networks. *IEEE Transactions on Computers*, 37:449–453, 1988.

[2] Ramón Béjar, Carmel Domshlak, Cèsar Fernández, Carla Gomes, Bhaskar Krishnamachari, Bart Selman, and Magda Valls. Sensor networks and distributed csp: communication, computation and complexity. *Artif. Intel.*, 161:117–147, 2005.

[3] Christian Bessiere and Jean-Charles Régin. MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems. In *Proceedings of CP'96*, pages 61–75, 1996.

[4] Christian Bessiere, Arnold Maestre, Ismel Brito, and Pedro Meseguer. Asynchronous backtracking without adding links: a new member in the ABT family. *Artif. Intel.*, 161:7–24, 2005.

[5] Ismel Brito and Pedro Meseguer. Synchronous, Asynchronous and Hybrid Algorithms for DisCSP. In *Proceedings of DCR'04*, pages 80–94, 2004.

[6] Ismel Brito and Pedro Meseguer. Improving ABT Performance by Adding Synchronization Points. In *Recent Advances in Constraints*, volume 5129 of *Lecture Notes in Computer Science*, pages 47–61. 2008.

[7] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Computer Systems*, 3(1):63–75, 1985.

[8] Anton Chechetka and Katia Sycara. A Decentralized Variable Ordering Method for Distributed Constraint Optimization. Technical Report CMU-RI-TR-05-18, Robotics Institute, Carnegie Mellon University, 2005.

[9] Anton Chechetka and Katia Sycara. No-Commitment Branch and Bound Search for Distributed Constraint Optimization. In *Proceedings of AAMAS'06*, pages 1427–1429, 2006.

[10] To-Yat Cheung. Graph Traversal Techniques and the Maximum Flow Problem in Distributed Computation. *IEEE Trans. on Soft. Engin.*, 9(4):504–512, 1983.

[11] Yek Loong Chong and Youssef Hamadi. Distributed Log-based Reconciliation. In *Proceedings of ECAI'06*, pages 108–112, 2006.

[12] Zeev Collin, Rina Dechter, and Shmuel Katz. On the feasibility of distributed constraint satisfaction. In *Proceedings of IJCAI'91*, pages 318–324, 1991.

[13] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artif. Intel.*, 41(3):273–312, 1990.

[14] Rina Dechter. Constraint Networks (survey). *In S. C. Shapiro (Eds.), Encyclopedia of Artificial Intelligence*, 1:276–285, 1992.

[15] Eugene C. Freuder and Michael J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proceedings of IJCAI'85*, pages 1076–1078, 1985.

[16] John Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the Second Canadian Conference on Artificial Intelligence,*, pages 268–277, 1978.

[17] Matthew L. Ginsberg. Dynamic Backtracking. *JAIR*, 1:25–46, 1993.

[18] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intel.*, 14(3):263–313, 1980.

[19] Katsutoshi Hirayama and Makoto Yokoo. The Effect of Nogood Learning in Distributed Constraint Satisfaction. In *Proceedings of ICDCS'00*, pages 169–177, 2000.

[20] Hyuckchul Jung, Milind Tambe, and Shriniwas Kulkarni. Argumentation as Distributed Constraint Satisfaction: Applications and Results. In *Proceedings of AGENTS'01*, pages 324–331, 2001.

[21] Thomas Léauté and Boi Faltings. Coordinating Logistics Operations with Privacy Guarantees. In *Proceedings of the IJCAI'11*, pages 2482–2487, 2011.

[22] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Series, 1997.

[23] Rajiv T. Maheswaran, Milind Tambe, Emma Bowring, Jonathan P. Pearce, and Pradeep Varakantham. Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling. In *Proceedings of AAMAS'04*, 2004.

[24] Amnon Meisels and Oz Lavee. Using additional information in DisCSP search. In *Proceedings of DCR'04*, 2004.

[25] Amnon Meisels and Roie Zivan. Asynchronous Forward-Checking for Distributed CSPs. In *Frontiers in Artificial Intelligence and Applications*, 2003.

[26] Amnon Meisels and Roie Zivan. Asynchronous Forward-checking for DisCSPs. *Constraints*, 12(1):131–150, 2007.

[27] Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe, and Makoto Yokoo. An Asynchronous Complete Method for Distributed Constraint Optimization. In *Proceedings of AAMAS'03*, pages 161–168, 2003.

[28] Viet Nguyen, Djamila Sam-Haroud, and Boi Faltings. Dynamic Distributed BackJumping. In *Recent Advances in Constraints*, volume 3419, pages 71–85. 2005.

[29] Adrian Petcu and Boi Faltings. A Value Ordering Heuristic for Distributed Resource Allocation. In *Proceedings of Joint Annual Workshop of ERCIM/CoLogNet on CSCLP'04*, pages 86–97, 2004.

[30] Adrian Petcu and Boi Faltings. DPOP: A Scalable Method for Multiagent Constraint Optimization. In *Proceedings of IJCAI'05*, pages 266–271, 2005.

[31] Adrian Petcu and Boi Faltings. ODPOP: An Algorithm for Open/Distributed Constraint Optimization. In *Proceedings of AAAI'06*, pages 703–708, 2006.

[32] Patrick Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9:268–299, 1993.

[33] Marius-Calin Silaghi. Generalized Dynamic Ordering for Asynchronous Backtracking on DisCSPs. In *Proceedings of DCR'06*, 2006.

[34] Marius-Calin Silaghi and Boi Faltings. Asynchronous aggregation and consistency in distributed constraint satisfaction. *Artif. Intel.*, 161:25–53, 2005.

[35] Mohamed Wahbi, Redouane Ezzahir, Christian Bessiere, and El-Houssine Bouyakhf. DisChoco 2: A Platform for Distributed Constraint Reasoning. In *Proceedings of DCR'11*, pages 112–121, 2011. URL http://www.lirmm.fr/coconut/dischoco/.

[36] Richard J. Wallace and Eugene C. Freuder. Constraint-Based Multi-Agent Meeting Scheduling: Effects of agent heterogeneity on performance and privacy loss. In *Proceedings of DCR'02*, pages 176–182, 2002.

[37] W. Yeoh, A. Felner, and S. Koeing. BnB-ADOPT: An Asynchronous Branch-and-Bound DCOP Algorithm. In *In Proceedings of Workshop on Distributed Constraint Reasoning (DCR'07).*, 2007.

[38] Makoto Yokoo. Algorithms for Distributed Constraint Satisfaction Problems: A Review. *Journal of AAMAS*, 3(2):185–207, 2000.

[39] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings of 12th IEEE Int'l Conf. Distributed Computing Systems*, pages 614–621, 1992.

[40] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Trans. on Knowledge and Data Engineering*, 10:673–685, 1998.

[41] Roie Zivan and Amnon Meisels. Synchronous vs Asynchronous Search on DisCSPs. In *Proceedings of EUMAS'03*, 2003.

[42] Roie Zivan and Amnon Meisels. Message delay and DisCSP search algorithms. *Annals of Mathematics and Artificial Intelligence*, 46(4):415–439, 2006.

[43] Roie Zivan and Amnon Meisels. Dynamic Ordering for Asynchronous Backtracking on DisCSPs. *Constraints*, 11(2-3):179–197, 2006.